MICROCHIP
WebSeminars

MPLAB® SIM

MPLAB IDE Software Simulation Engine

 MPLAB SIM Software Simulation Engine Slide 1

Welcome to this web seminar on **MPLAB SIM**, the software simulator that comes with the free **MPLAB Integrated Development Environment**, or **IDE**.

My name is **Darrel Johansen** and I'm a manager for Development Systems at Microchip Technology.

This 20 minute seminar will focus on **MPLAB SIM**, and will demonstrate how this tool can be used to develop and debug code for Microchip microcontrollers.

MICROCHIP WebSeminars

MPLAB® SIM WebSeminar Agenda

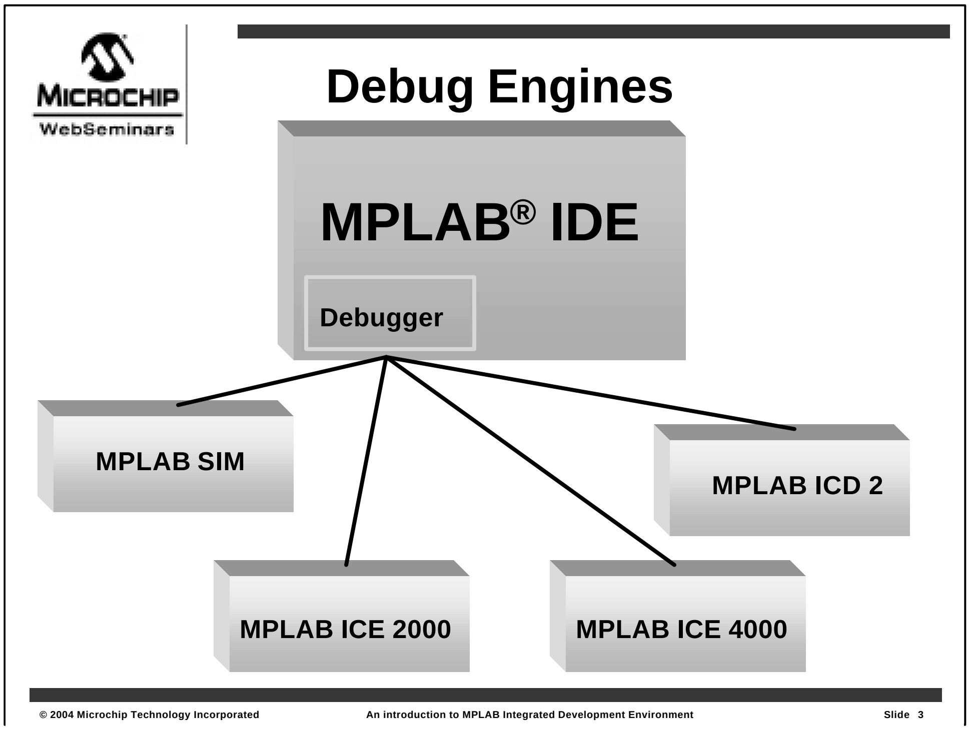
- MPLAB SIM is one “debug engine” used by MPLAB IDE
- MPLAB SIM has debug features similar to other debug engines
- MPLAB SIM has some unique debug features
- MPLAB SIM can have input stimulus signals and can log register outputs to files

This seminar will describe MPLAB SIM, one of the **debug engines** available for MPLAB IDE.

It has similar features to other debug engines, allowing the engineer to switch from one to another without encountering a new learning curve.

MPLAB SIM also has some unique debugging features that are not available in the hardware debuggers.

In order to debug the operation of code with simulated external signals, input stimulus events can be created. And in order to measure how an application performs, registers can be logged to files, graphed and be subjected to further analysis.



The other debug engines are **hardware** devices, while MPLAB SIM is a **software** program, running on your **PC**.

MPLAB SIM provides many of the same features as in-circuit emulators and in-circuit debuggers. The **difference** is that both in-circuit emulators and in-circuit debuggers allow the code to be run on actual **silicon**, and also allow a target application hardware to be functional while being debugged.

MPLAB SIM has features to simulate hardware interaction with other signals and devices, and since it is running as software on the PC, it has **complete information** about the internal **state** of the simulated chip at **each instruction**. This is a little different from the hardware debuggers because, while they are running code at full speed, they typically cannot monitor all registers and all memory in real time.

Both MPLAB SIM and the hardware debuggers can do the traditional functions of debuggers, but due to their differences, they can have unique features of their own. This presentation will identify the functions and features of MPLAB SIM.

Debugger

Basic Functions:

- Reset Target
- Execute Code
- Halt Code
- Examine/Modify Registers/Memory
- Single-Step Code

 An introduction to MPLAB Integrated Development Environment Slide 4

The debugger is a part of MPLAB IDE, and whether you are using MPLAB SIM, MPLAB ICE or MPLAB ICD 2, most operations are exactly the same. This allows you to develop code using the simulator, then when your hardware is ready, you can use a hardware debugger to further test your code in practice without having to learn how to use a new tool.

These are the basic debug functions:

•**Reset** the target, in order to restart the application

•**Execute** the code so the program can be tested to verify it functions as designed

•**Halt** the code at breakpoints

•While halted at breakpoints, memory and variables can be **examined and modified** to analyze and debug the application code

•To closely inspect how code executes, each instruction can be **Single stepped**. This allows the engineer to go through code one instruction at a time while monitoring affected variables, registers and flags. Single stepping essentially “zooms in” on code to ensure that it operates properly in complex and critical sections with ranges of variable values and under various test conditions.

Debugger

Advanced Functions:

- Watch Points
- Trace Buffer
- Stopwatch
- Complex Breakpoints
- Correlate Machine Code Execution and Memory Contents with High Level Language Source

 An introduction to MPLAB Integrated Development Environment Slide 5

Most debuggers also have additional features to help analyze and debug the application. Some of these are listed here:

• **Watch points** group and monitor selected variables and memory locations into a convenient, custom readout.

• **Trace buffers** capture the streams of instructions executed and reveal the contents of changing register values.

• A **Stopwatch** can time a section of code. Routines can be optimized, and critical code timing can be accurately measured and adjusted.

• **Complex breakpoints** offer a method for establishing breakpoints or for gathering data in the trace buffer based upon **multiple conditions**. Simple breakpoints allow setting breakpoints in the source code or anywhere in program memory. Complex breakpoints allow getting a breakpoint on a condition such as,

- "**after** the main routine called "**RefreshDisplay**" executes then
- **wait** for subroutine "**ReadTemp**" to execute. Then
- **break** if the variable named "**Temperature**" is greater than **20**. "

Complex events can even be constructed to count events, so that a subroutine would have to be executed 15 times before it starts looking for a value on a pin or in a register. This kind of breakpoint allows you to describe the **condition** where your code misbehaves, then gets a breakpoint or traces code **at that condition**. This is usually a faster way of finding bugs than simply setting simple breakpoints and stepping through your code.

• Finally, most advanced debuggers allow you to **correlate** the execution of the application on the target with the source code. This allows you to single step through C source code, even though each C statement may generate many lines of machine code. Likewise, memory storage in file registers is correlated with the variables you use in your program. So if you have a floating point number that spans multiple machine file registers, you can monitor the multiple file register contents in a Watch Point to display the value in floating point representation.

MICROCHIP WebSeminars

Simulator

- Runs as a software program on PC
- Runs at a speed determined by PC speed, simulation activities, operating system tasks, but times events based upon simulated operating frequency.
- Simulates core CPU, memory, many peripherals
- Responds to simulated inputs called stimulus signals.
- Can log outputs to files for further analysis.

 An introduction to MPLAB Integrated Development Environment Slide 6

MPLAB SIM is a simulator, and as a result it has certain characteristics that make it a unique debug engine. In modern Windows based PCs, many things are happening in the background --other programs may be running, hardware may be communicating to the PC, and so on. So the speed of the simulation is determined by

•how **fast** your PC executes,

•the **complexity** of the current simulation, and

•the **number of other tasks** executing on your PC.

Currently the maximum speed of MPLAB SIM is on the order of **10 MIPS**, or 10 million instructions per second. This will be affected by how many other things are being done by your PC, by the code the simulator is running, and by the other tasks that the simulator is performing.

The simulator simulates the operation of

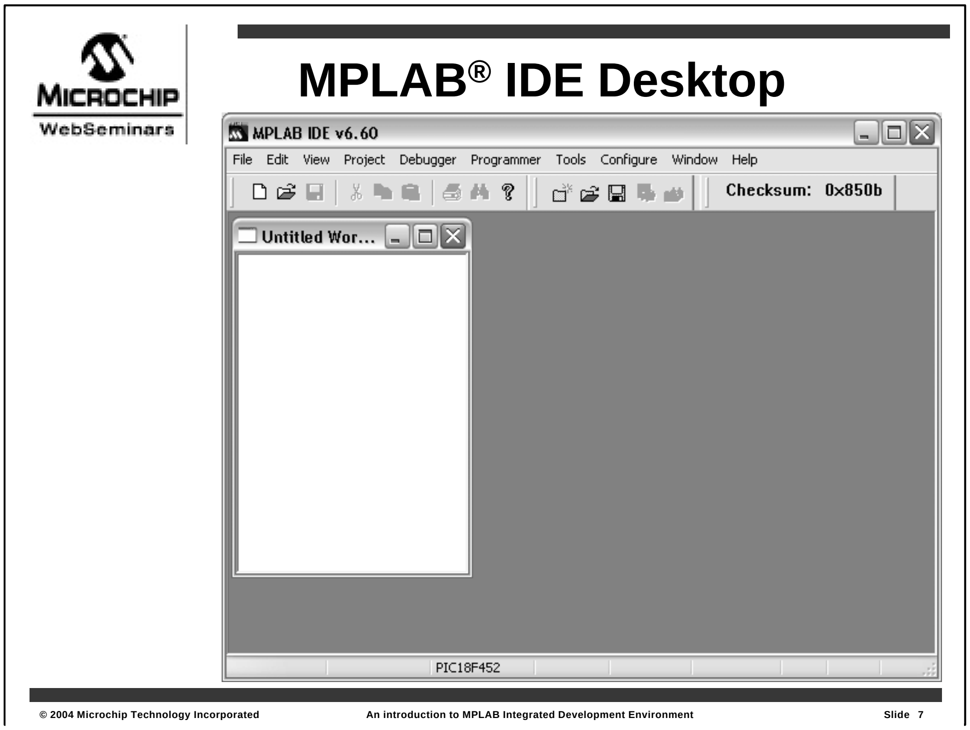
•the core **CPU** and it's internal registers,

•**memory**, and

•many of it's **peripherals**.

In order to test the application on the simulator, **stimulus** signals can be applied to pins and registers.

To evaluate performance, the simulator can **log** changing registers to files for further analysis.



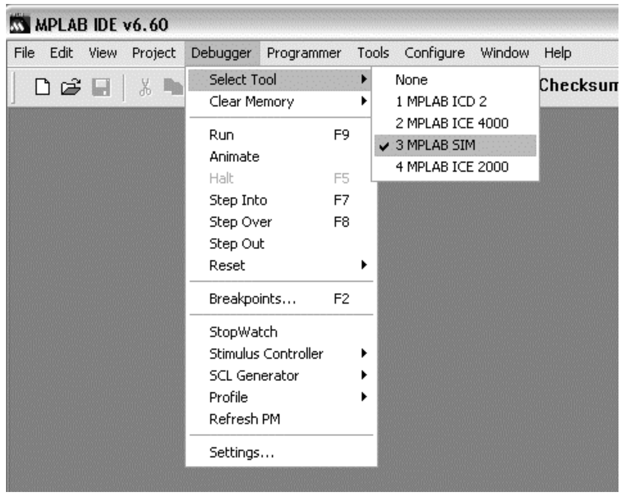
This is the MPLAB IDE desktop when it is first started up.

There is

- a standard **menu** across the top
- a **tool bar** below this,
- a blank "**Workspace window**," and
- a **status bar** on the bottom.

MICROCHIP WebSeminars

MPLAB® IDE Select Debug Engine

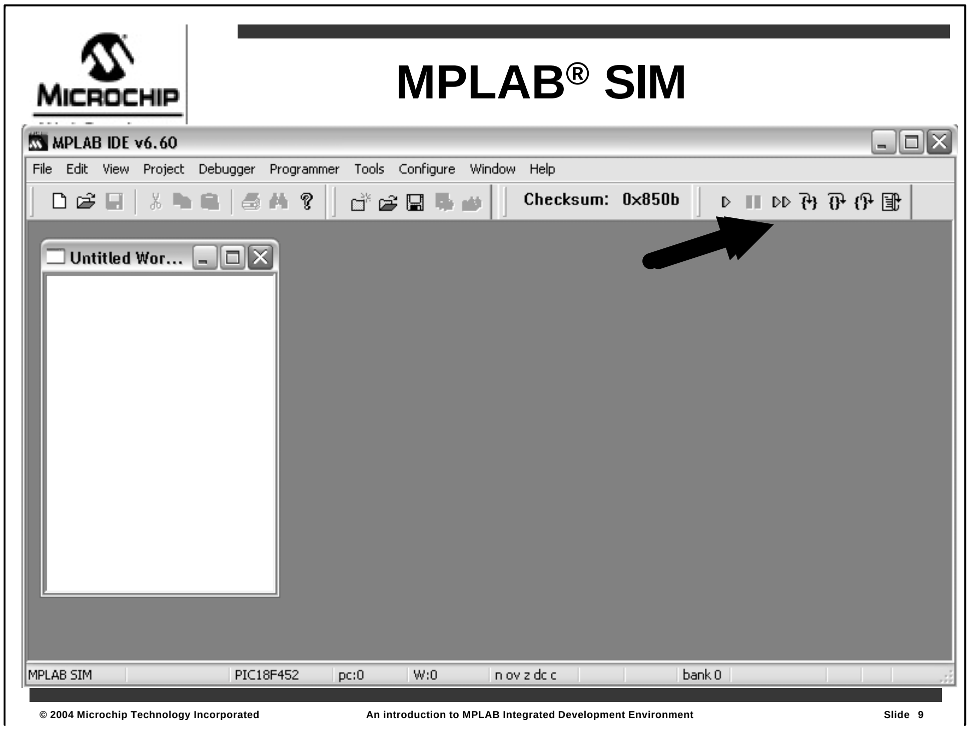


 An introduction to MPLAB Integrated Development Environment Slide 8

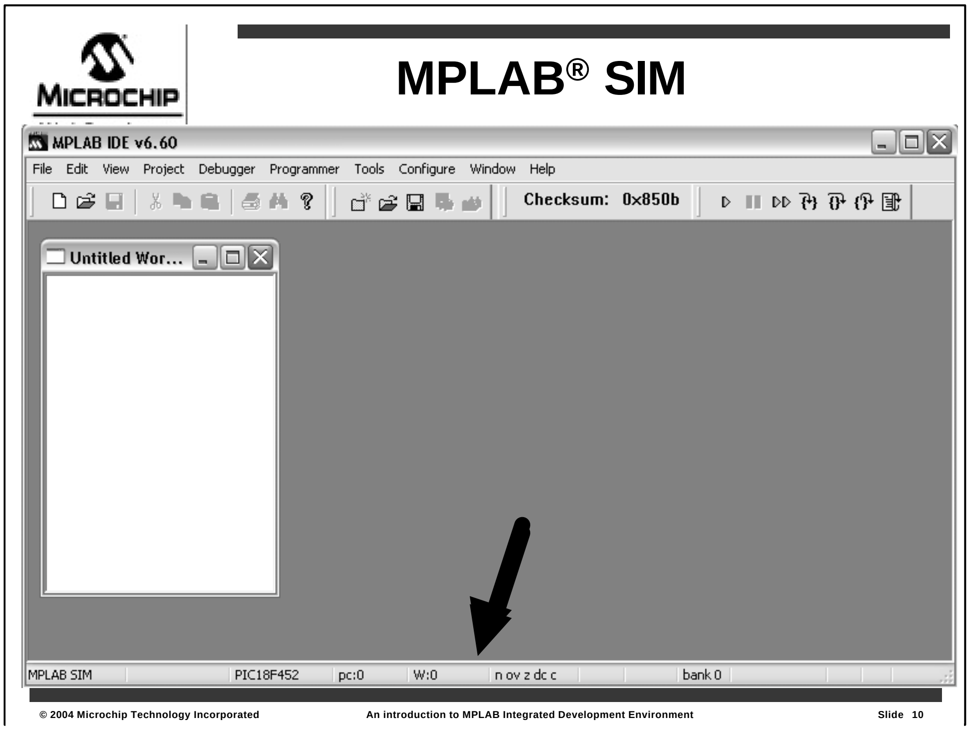
MPLAB SIM is selected as our debug engine from the Debugger menu.

Note the other functions on the debug menu, such as **Run**, **Step**, and **Breakpoints**.

We’ll look closer at some of these other options soon.

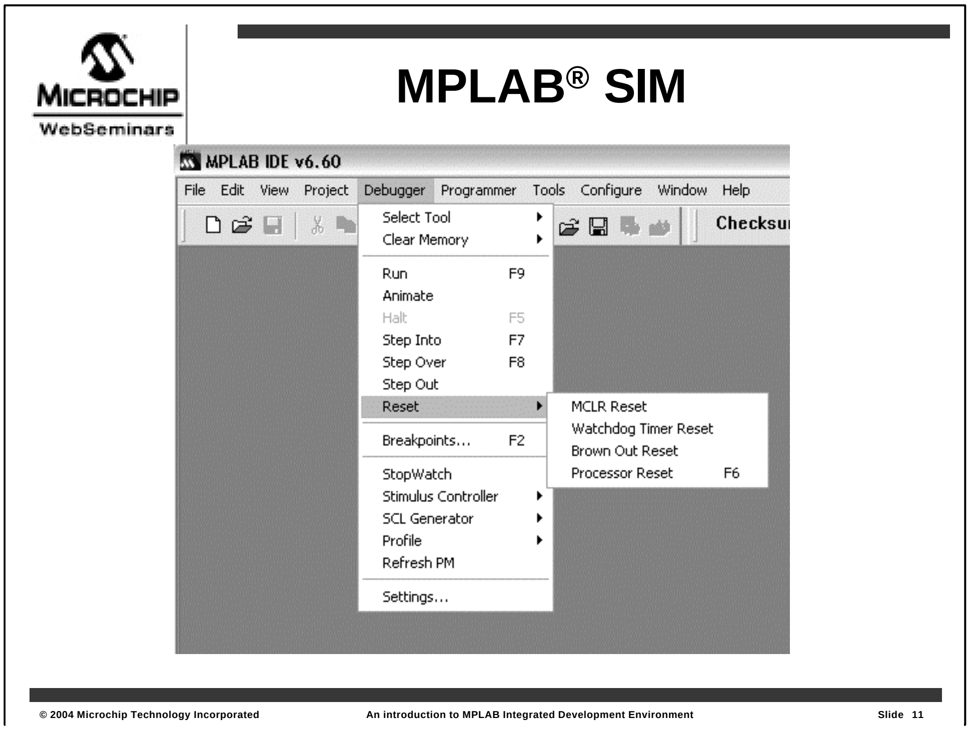


Once a debug engine is selected, the toolbar is appended with some new icons for running, halting, single stepping, and resetting the target.



The Status bar **now** shows some additional information as well.

- **MPLAB SIM** shows as our current debug engine.
- The simulated processor is listed, in this case the **PIC18F452**,
- then the **program counter**,
- the **W register**,
- the current state of the **internal CPU flags** and
- the current selected **file register bank**.



Operations can be selected in multiple ways depending upon how you like to work. You can select functions from:

•The **toolbar icons**,

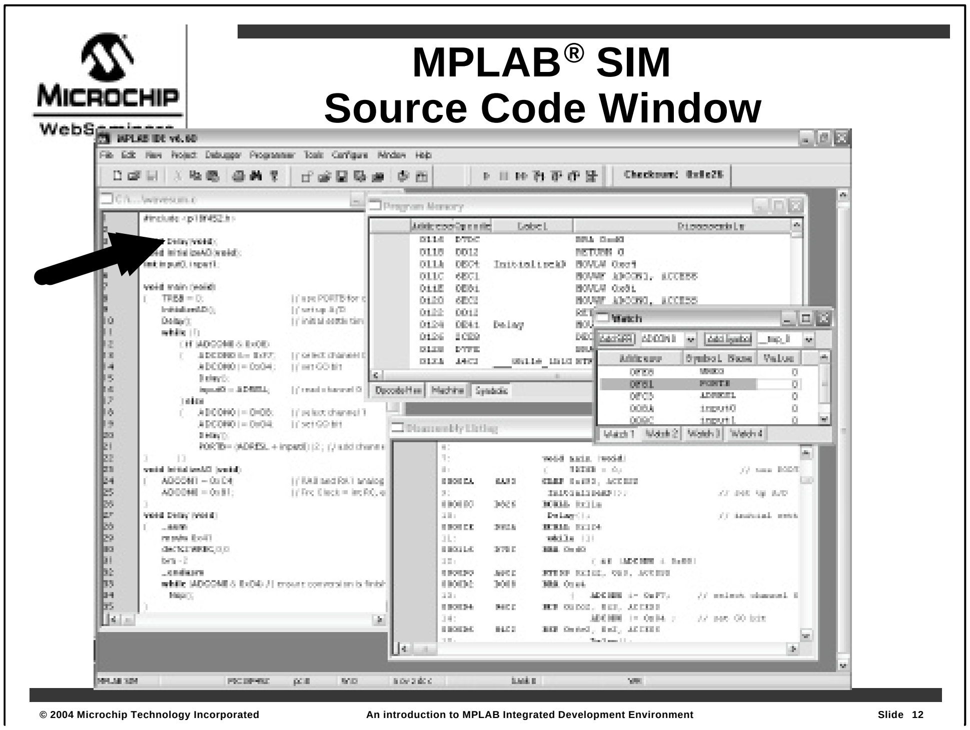
•the **menus**, or

•the **hot keys** listed on the menus

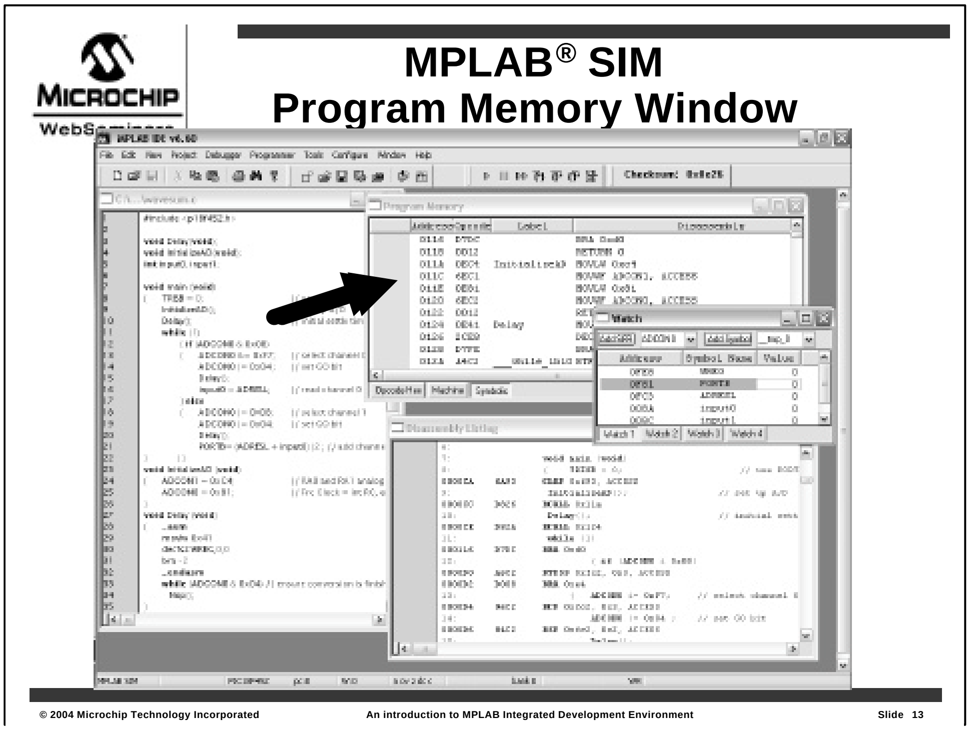
can be used to execute the debug functions. Note that some functions are a little more complex, such as Reset, which actually has four types of reset actions.

Once MPLAB SIM is established as the debug engine, whenever a project is built, it is automatically loaded into the simulator’s **program memory** to be run and tested.

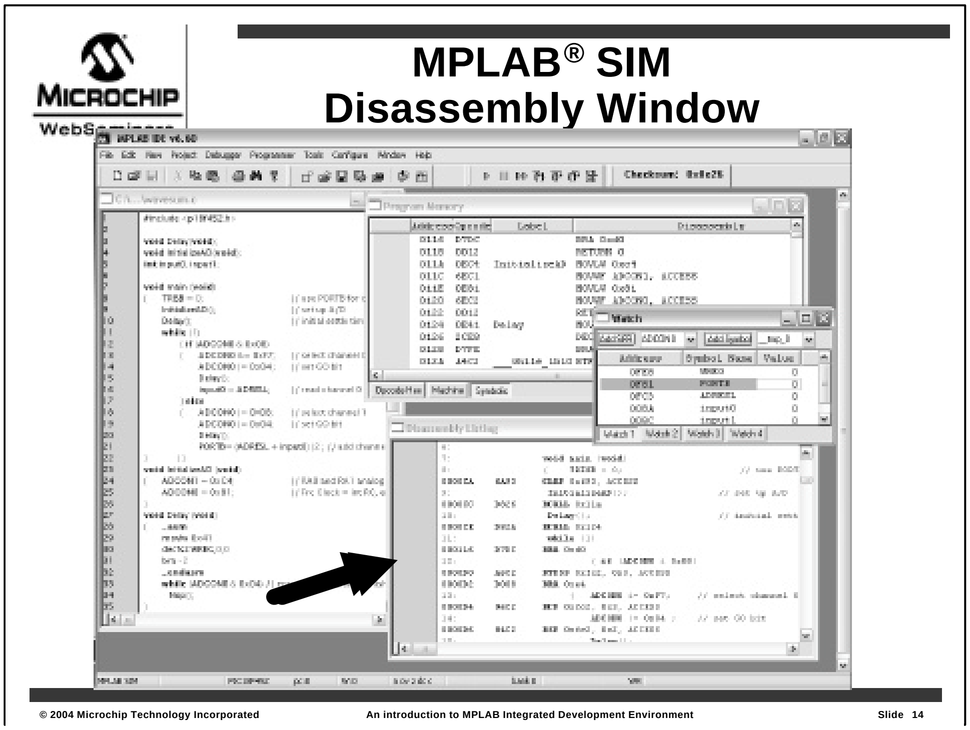
Various debug windows become available…<click>



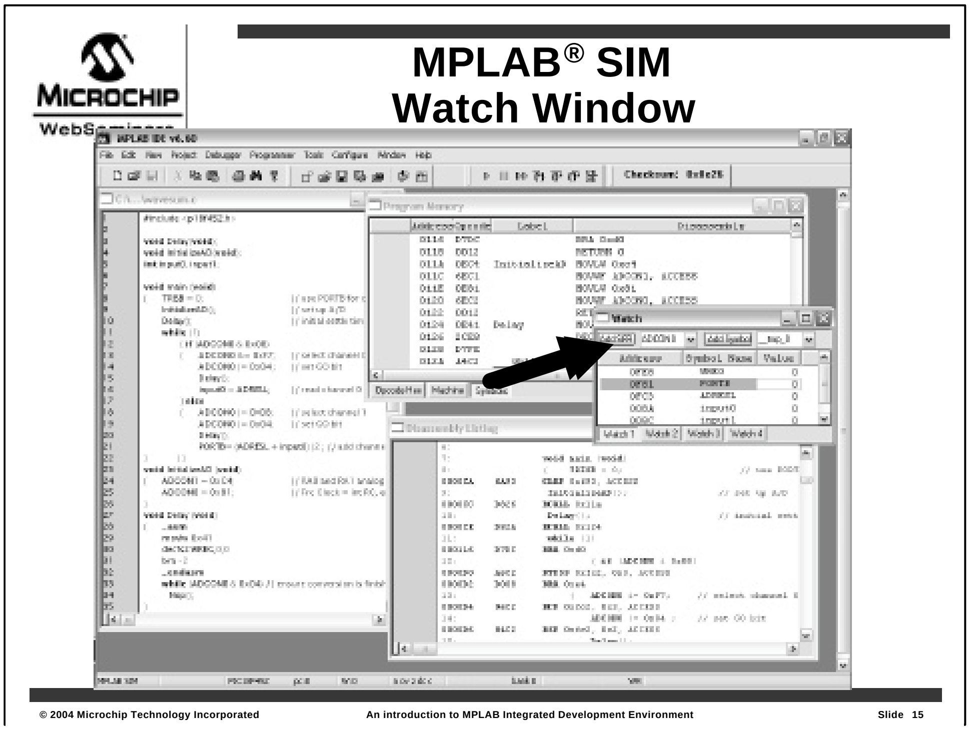
One debug window is the **source code window**. This is actually the **editor**, and breakpoints can be set by clicking on a line with the right mouse button. Single stepping with the source code window in focus will single step through the **C source lines**. Since you are in the editor, changes can be done quickly, and the project can be rebuilt.



The **Program Memory window** shows the machine code that will be executed by the simulator. Single stepping with this window in focus will allow you to step through each **machine instruction**.



Another window, called the **Disassembly Listing window** shows high level **source code interspersed with machine code** generated by each C statement.

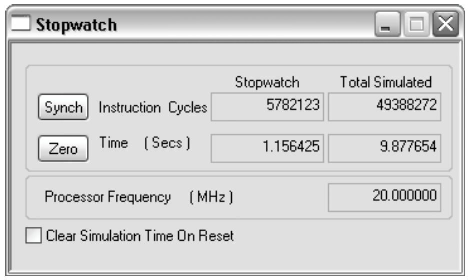


You can also open up a **watch window** and drag the variables from your program to see the contents as you break and single-step through your code.

While debugging, other windows are available to view

- •**register** memory,
- •**stack** memory, and
- • non-volatile **data memory** areas.

MPLAB® SIM
Stopwatch



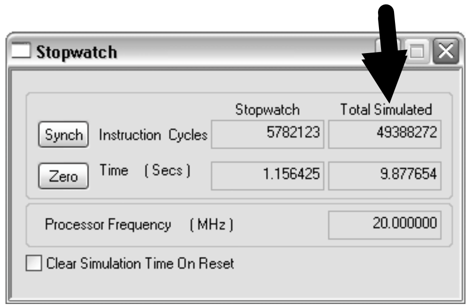
The **Stopwatch dialog** can time sections of code as they run on the simulator.

The Stopwatch calculations are based upon the instructions executed and the setting entered for the Processor Frequency. The processor frequency is set to 20 Mhz in this example.

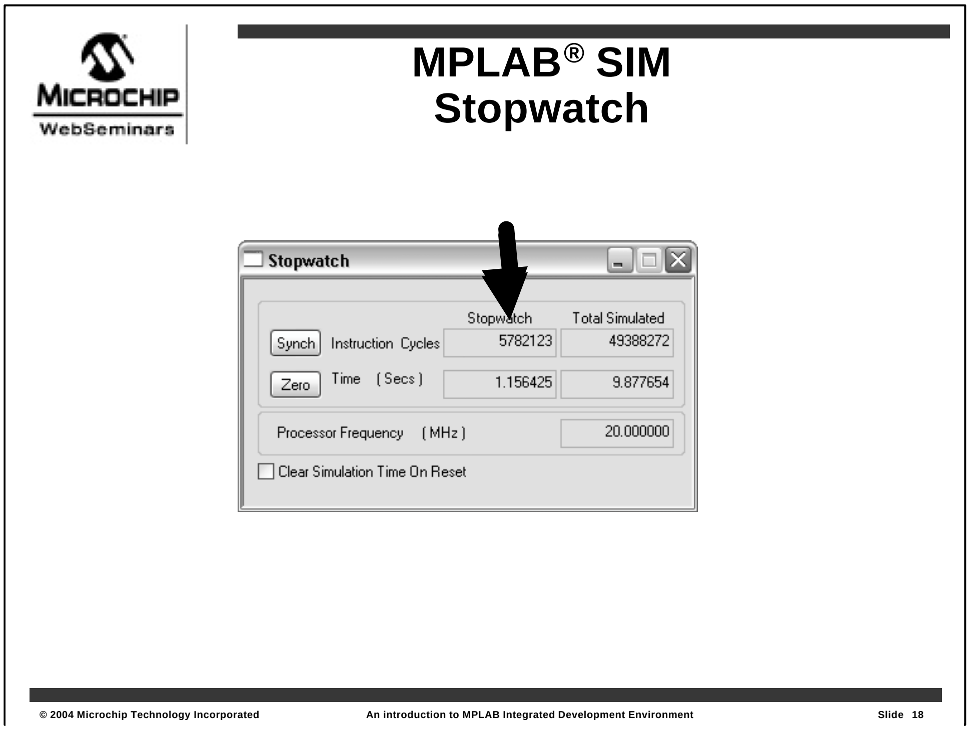
From the number of instruction cycles executed, the total time is calculated.

This is the time it would take to run on a real chip.

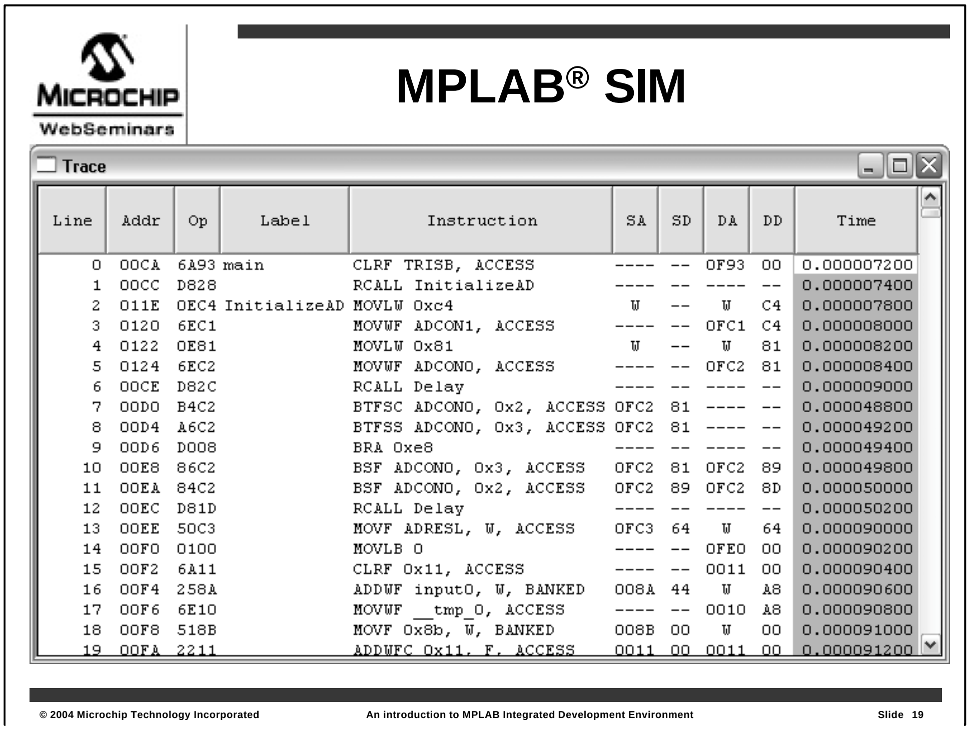
MPLAB® SIM Stopwatch



The stopwatch has two pairs of readouts, one tells the **total** simulated clock cycles and the corresponding execution time…<click>



…and the other can be zeroed out, to make a **measurement** from one breakpoint to the next.



Line	Addr	Op	Label	Instruction	SA	SD	DA	DD	Time
0	00CA	6A93	main	CLRF TRISB, ACCESS	----	--	0F93	00	0.000007200
1	00CC	D828		RCALL InitializeAD	----	--	----	--	0.000007400
2	011E	0EC4	InitializeAD	MOVLW 0xc4	W	--	W	C4	0.000007800
3	0120	6EC1		MOVWF ADCON1, ACCESS	----	--	0FC1	C4	0.000008000
4	0122	0E81		MOVLW 0x81	W	--	W	81	0.000008200
5	0124	6EC2		MOVWF ADCON0, ACCESS	----	--	0FC2	81	0.000008400
6	00CE	D82C		RCALL Delay	----	--	----	--	0.000009000
7	00D0	B4C2		BTFSC ADCON0, 0x2, ACCESS	0FC2	81	----	--	0.000048800
8	00D4	A6C2		BTFSS ADCON0, 0x3, ACCESS	0FC2	81	----	--	0.000049200
9	00D6	D008		BRA 0xe8	----	--	----	--	0.000049400
10	00E8	86C2		BSF ADCON0, 0x3, ACCESS	0FC2	81	0FC2	89	0.000049800
11	00EA	84C2		BSF ADCON0, 0x2, ACCESS	0FC2	89	0FC2	8D	0.000050000
12	00EC	D81D		RCALL Delay	----	--	----	--	0.000050200
13	00EE	50C3		MOVF ADRESL, W, ACCESS	0FC3	64	W	64	0.000090000
14	00F0	0100		MOVLB 0	----	--	0FE0	00	0.000090200
15	00F2	6A11		CLRF 0x11, ACCESS	----	--	0011	00	0.000090400
16	00F4	258A		ADDWF input0, W, BANKED	008A	44	W	A8	0.000090600
17	00F6	6E10		MOVWF __tmp_0, ACCESS	----	--	0010	A8	0.000090800
18	00F8	518B		MOVF 0x8b, W, BANKED	008B	00	W	00	0.000091000
19	00FA	2211		ADDWFC 0x11, F, ACCESS	0011	00	0011	00	0.000091200

 An introduction to MPLAB Integrated Development Environment Slide 19

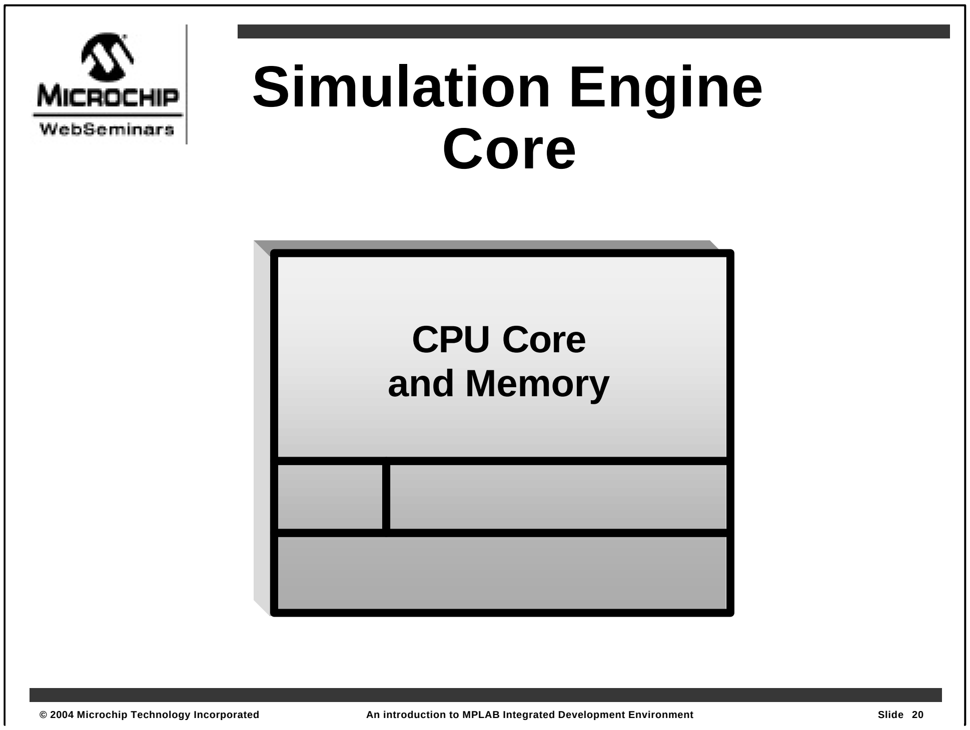
The stopwatch is one way to measure time in the simulator, but there is another:

The **trace buffer** records instructions when they execute and puts a **time stamp** on each instruction.

After you capture events in the trace buffer, you can time them.

The trace buffer has the advantage that it can capture large amounts of data selectively and each instruction has a time stamp.

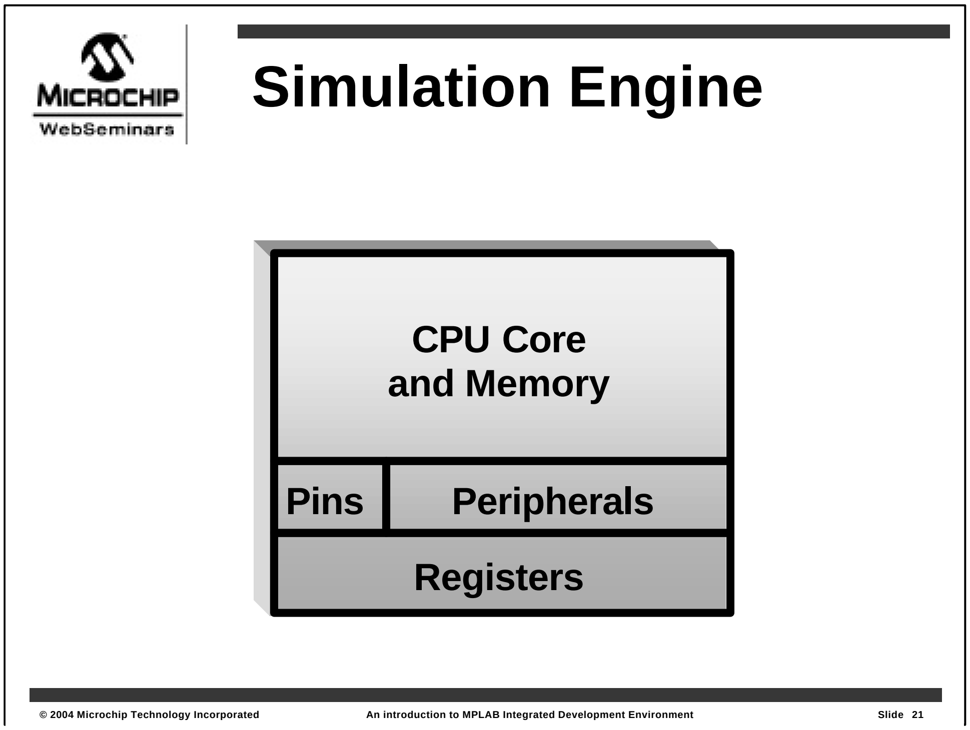
You can capture an interrupt routine, for instance, and then easily calculate the time **between interrupts** and the total time **each interrupt took** to execute.



A **block diagram** of the simulator might look like this. At the center is the simulation of

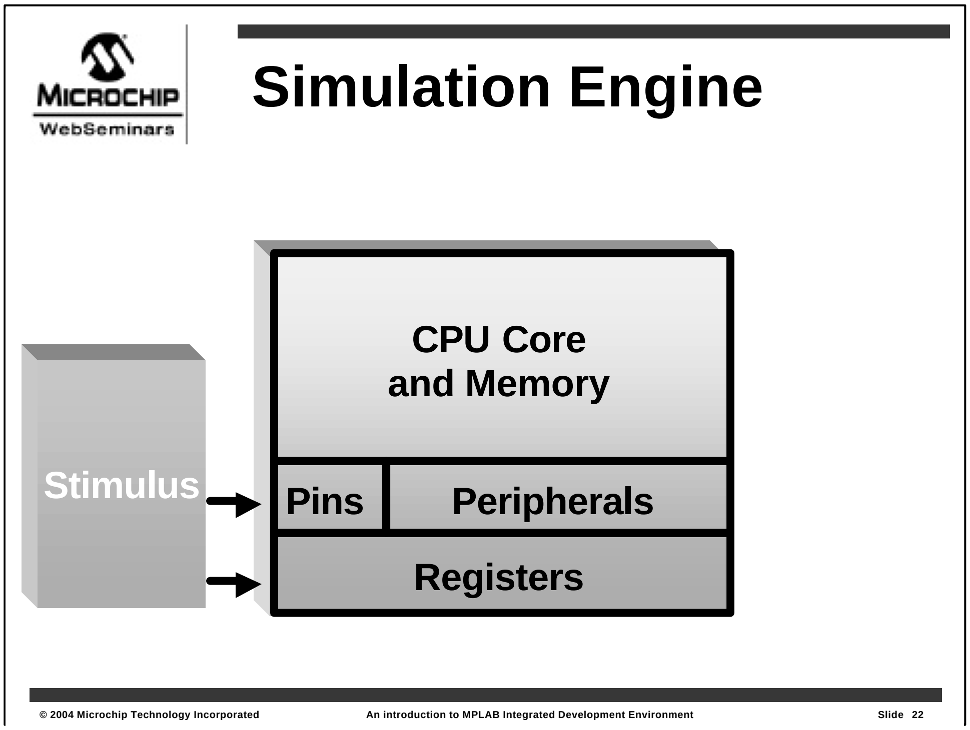
the CPU core with the various program, file and data memory areas;

- the **instructions**;
- the **stack**;
- the **program counter** and
- the **status flags** of the device being simulated.



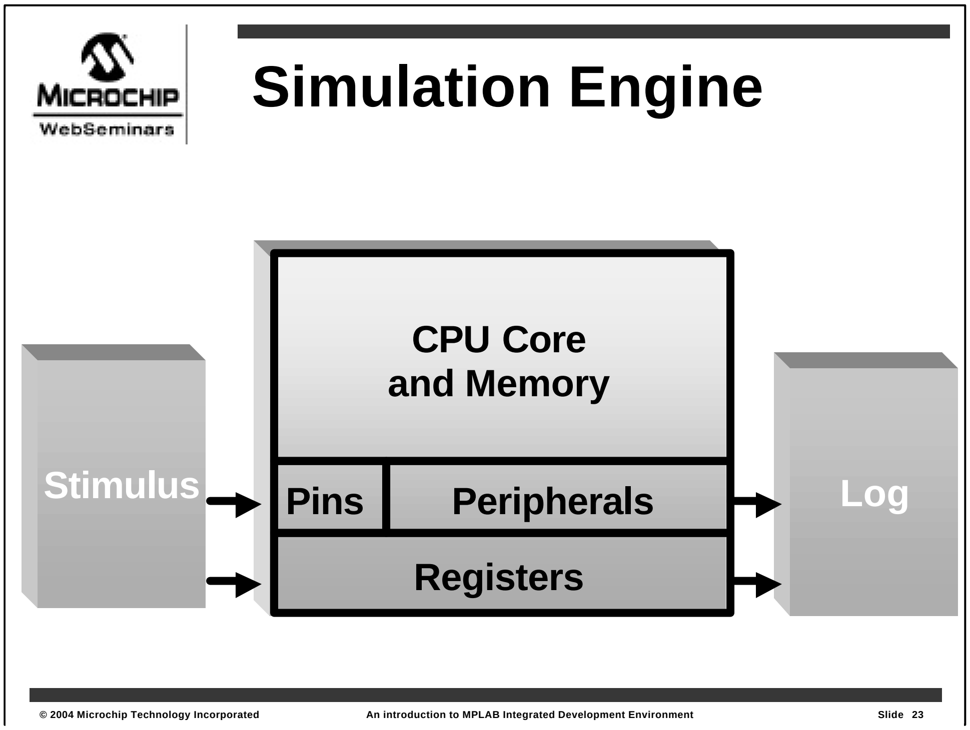
Simulation includes **pin inputs** and **outputs** as well as many of the other **peripherals**.

The peripherals communicate to the application through **special function registers**.



A complex **Stimulus Generator** simulates signals that can be applied to the device under simulation.

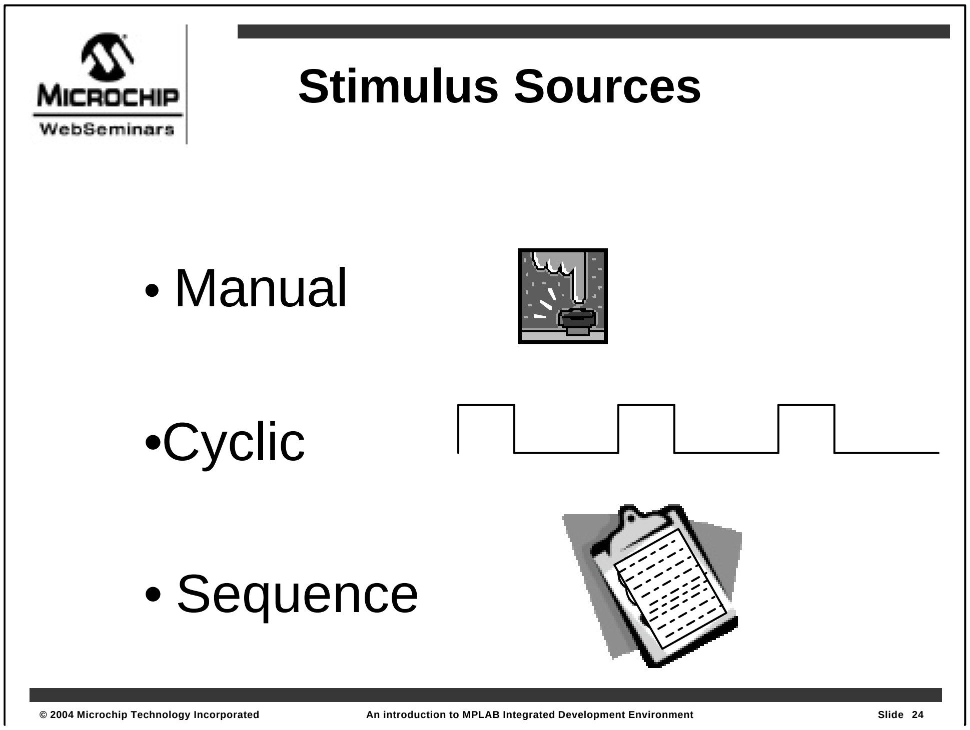
The stimulus generator can send signals to pins or to registers in the simulator.



The activity of the simulator can be sent to a **Log file** for later analysis. This is done by using either the **USART** as a communication device for inputs and outputs, or by using the **register log** feature. Both stimulus and logging activity can be driven either by

- execution at a specified **program counter address** or
- by sequencing "**on demand**."

"**On demand**" means that whenever that register is read by an instruction, a value is read from or written to a list, then advanced to the next position in the list.

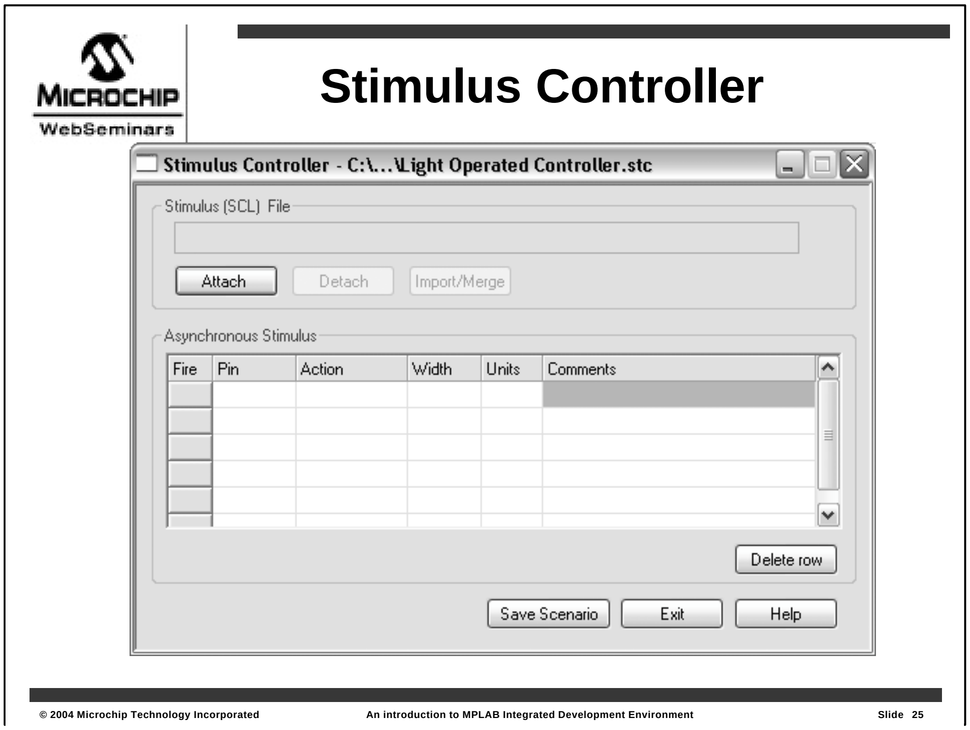


There are three types of Stimulus sources for MPLAB SIM:

- •**Manual triggers** are changes in digital signal levels caused by clicking on a button with a mouse. These allow you to simulate the action of closing a switch, or pulsing a pin.
- •A **Cyclic stimulus** generates a repeating waveform, either a for a pre-determined length of time, or continuously.
- •**Sequential data** is data that can be applied to pins, registers, or bits in registers from a list.

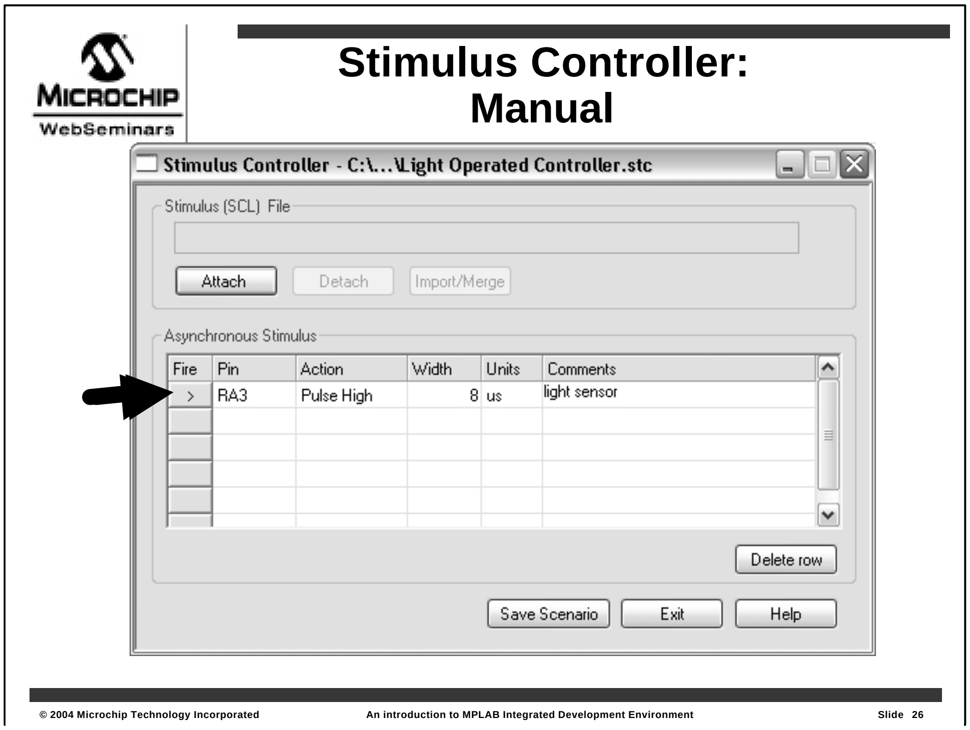
A list for sequential data can be entered in a

- •**dialog** or it can come from
- •a **file**.



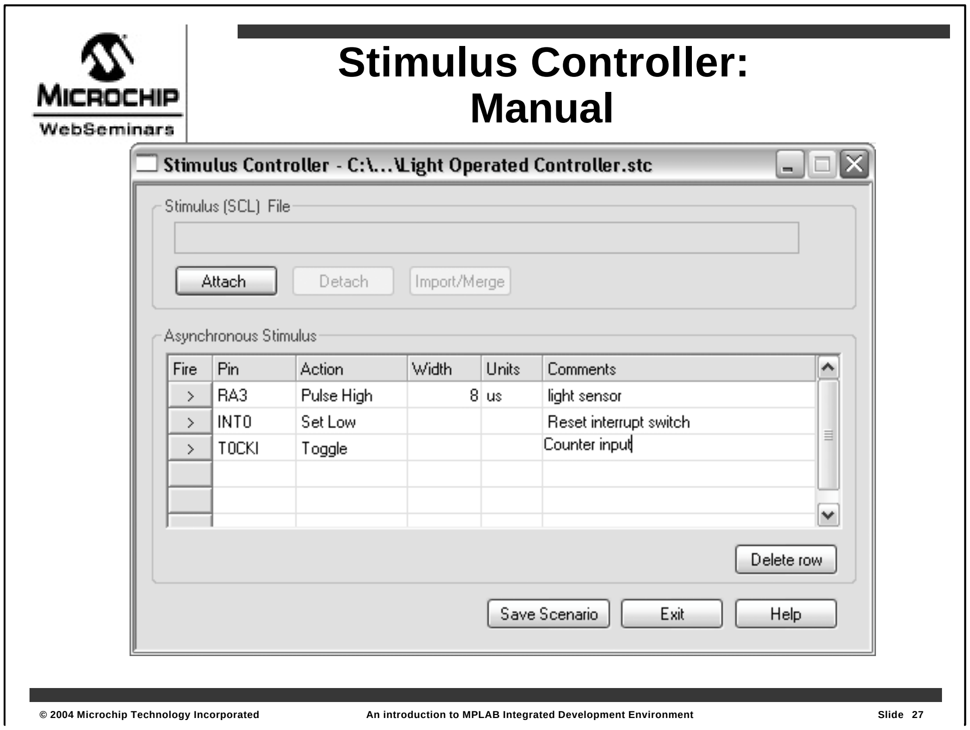
Here is the **Stimulus Controller** for MPLAB SIM.

In this dialog, you can select **actions** to apply to a **pin** and then, when your program is running, you can press the associated "**Fire**" button to the left of the pin name to **activate** that signal.



For instance, we could set up a pulse to occur on **PORTA** pin **RA3**.

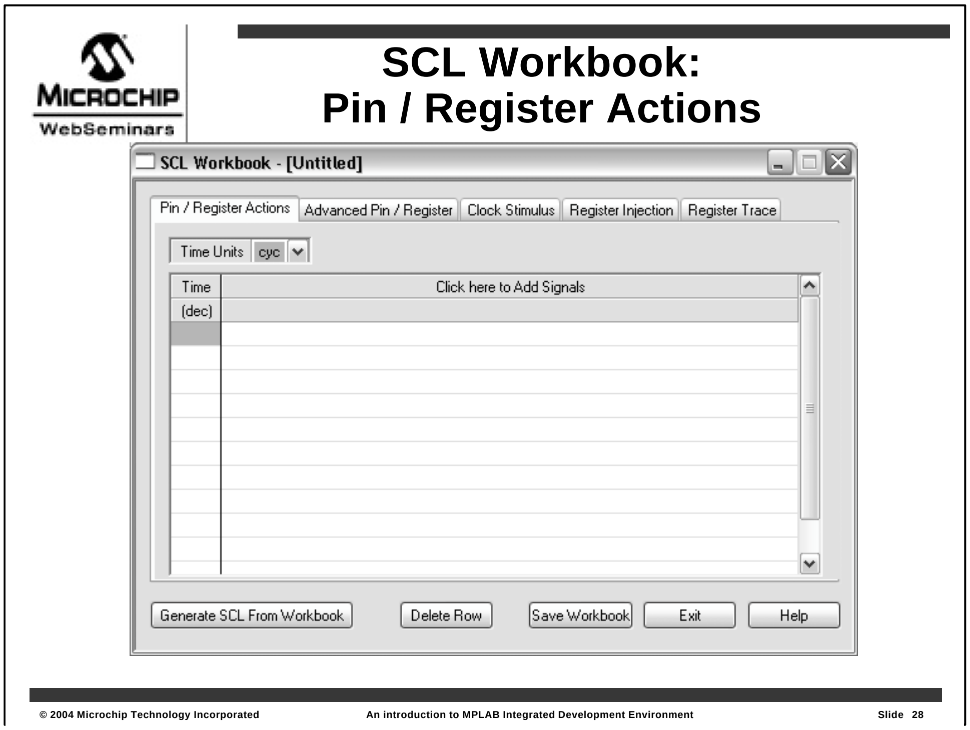
Here, when we press the **Fire** button, the simulated signal on **RA3** will **pulse high** for **8 microseconds**.



More actions can be added. Here two more are shown,

- one to force the **INT0 pin low** when we press the fire button, and
- one to toggle the **INPUT** pin for the Timer/Counter.

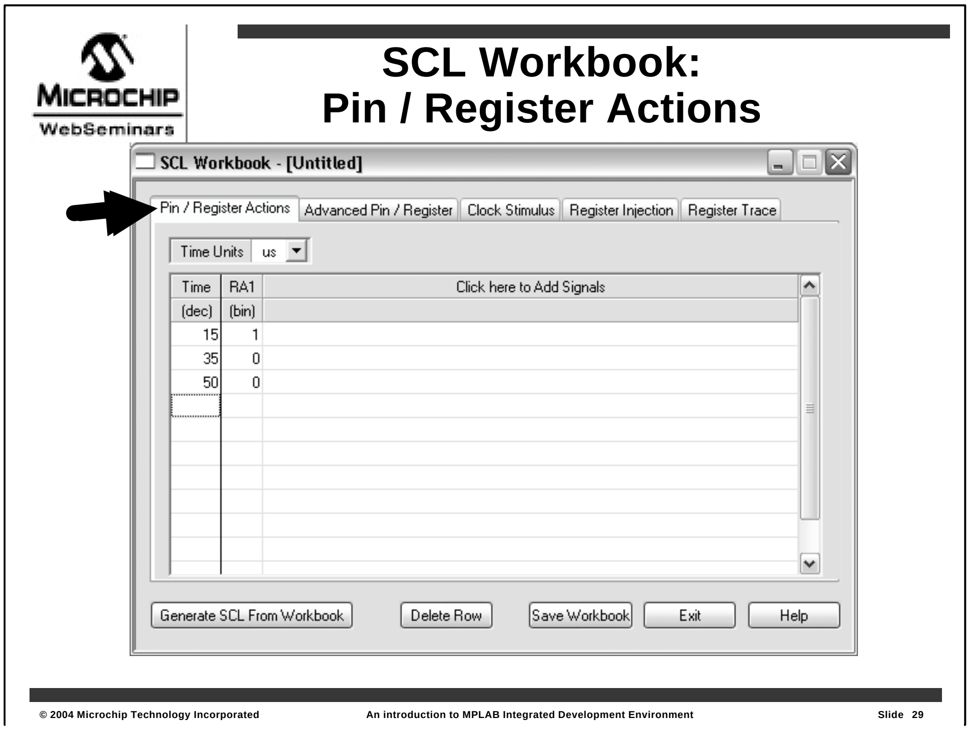
A “**Toggle**” Action will **alternate** between setting the pin **high** and **low** each time the Fire button is pressed.



For more complex stimulus control, actions are entered in the **SCL Workbook**.

SCL stands for **Stimulus Control Language**, but with the graphical user interface of MPLAB, all stimulus events can be set up with these easy graphical dialogs. Once the events are described in the SCL Workbook, they are compiled into an SCL file that can then be loaded into the Stimulus Controller.

There are **five tabs** in this workbook.

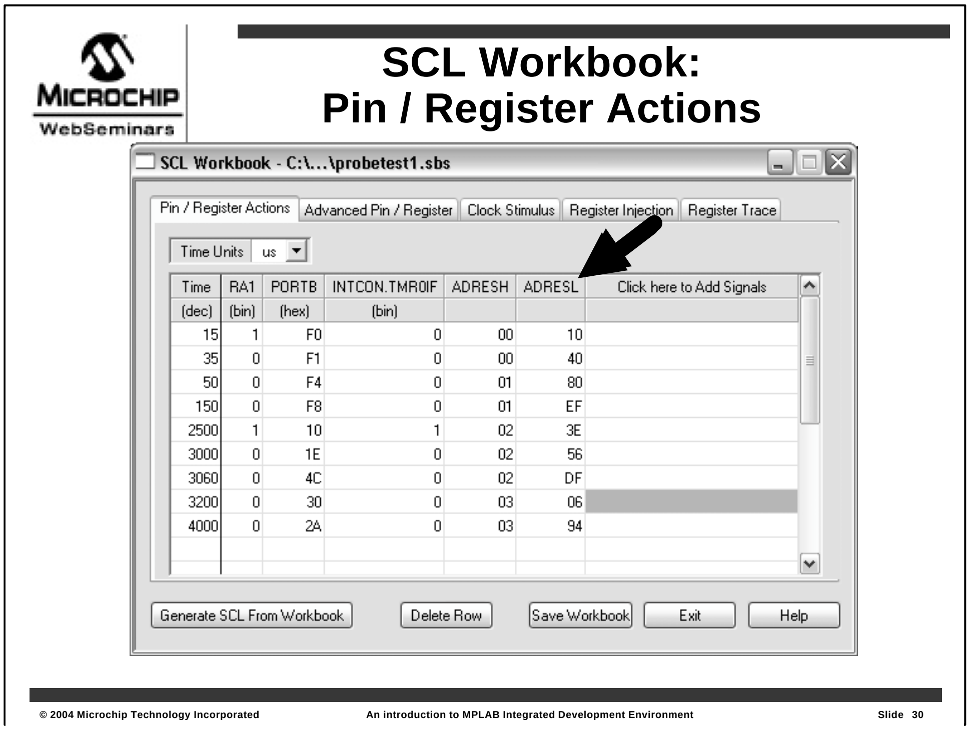


The **first tab** in the SCL Workbook uses a **list of times** and **signals** to be applied to **pins** and **registers**.

For instance, we can set up a series of events to happen on pin RA1.

- Here **15 microseconds** after the program is reset and **started**,
- the pin **RA1** will go **high**.
- Then at **35 microseconds** after the start of the program pin **RA1** will go **low**.

We can add other steps and signals to this dialog.



Here sequences of events are applied to

- •pin **RA1**,
- •**PORTB**,
- •the **Timer interrupt flag**, and
- •to the **A/D buffer**.

The list of values under the column labeled **PORTB** are **8-bit values** that will be applied to all **eight pins** of PORTB at the times entered down the left column.

The next column is a list for a signal connected to the

- •**Timer0 interrupt flag**,

an internal register.

The two columns on the right allow the

- •**A/D buffer**

to be set to a sequence of 10-bit values.

The second tab in the SCL Workbook, the **Advanced Pin/Register** tab, provides **conditional control** over events.

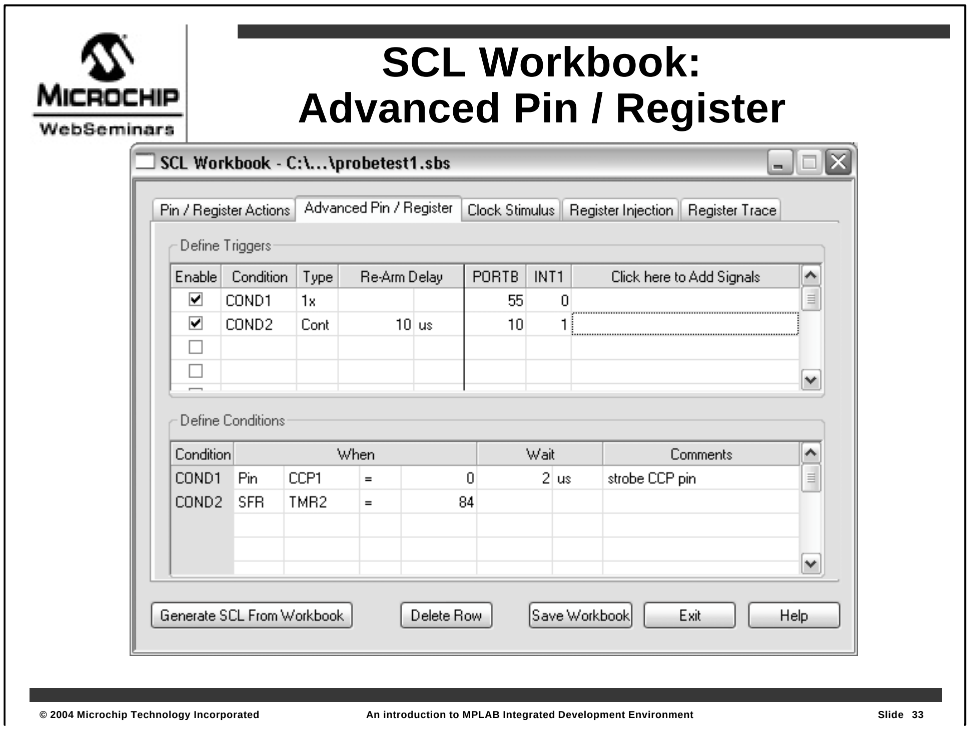
Here a **condition** must occur before the stimulus is activated.

The condition is described in the bottom section.

The condition is gating the event to happen

- **2 microseconds** after pin **CCP1** goes **low**.

When that occurs, a value of **55** will be applied to **PORTB**.



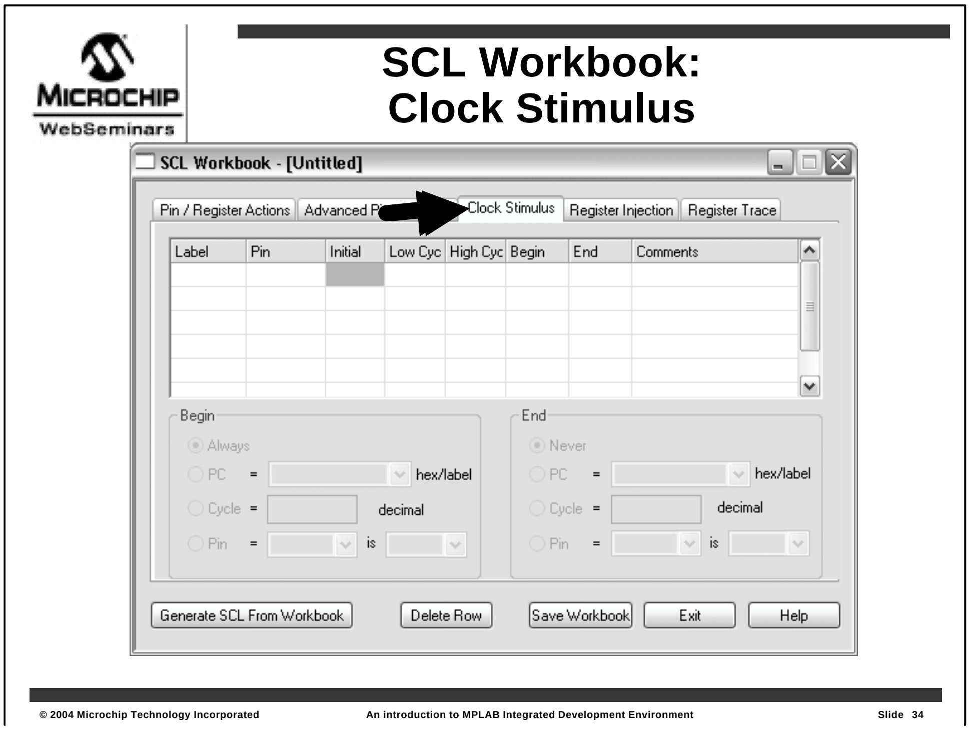
Other events can be described on subsequent lines.

The event can be a "**1x**" event, which means that it will occur only **once**, or it can be a **Continuous** event, meaning that will happen **every time** the associated condition occurs.

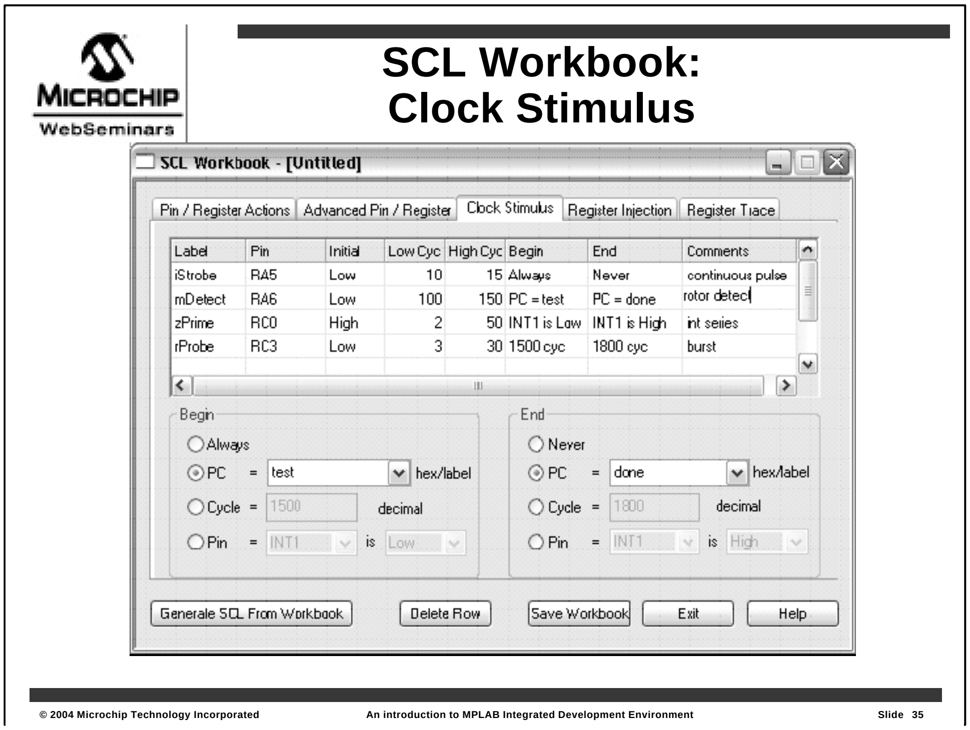
Here a second event happens each time **TMR2** reaches a value of **84**.

When this happens, **PORTB** will change to a value of **10**, and the **INT1** pin will go **high**.

There is a 10 microsecond **re-arm delay**. This prevents this event from happening **sooner** than 10 microseconds after its last firing.



The **Clock Stimulus** tab of the SCL Workbook generates **repeating digital waveforms**.



Four waveforms are entered here.

The first called **iStrobe**

- •holds the **RA5** pin **low** for **10 instruction cycles**, then
- •goes high for **15 instruction cycles**.
- •It starts **immediately**, and **never stops**.

A **Comments** column and the **Label** column are user-defined labels for this dialog, and are free-form, allowing any text entry. The label is arbitrary, simply for tagging the lines in this dialog with a descriptive name.

Line two describes a waveform that will be applied to **RA6**.

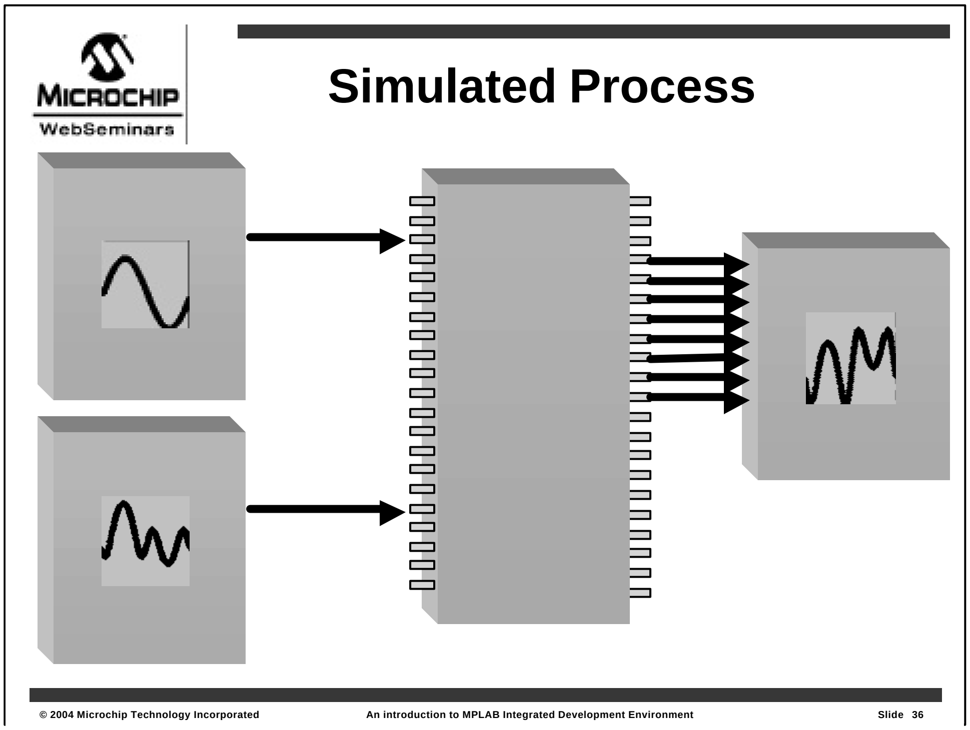
- •**RA6** will start out **low**,
- •stay **low** for **100 cycles**,
- •then go **high** for **150 cycles**.
- •It will **not start until** the program counter reaches the routine called "**test**."
- •It will begin cycling and
- •will **stop** when the program counter reaches a routine labeled "**done**."

Line three describes a stimulus event depending upon the **INT1** level**.** When INT1 goes low the signal on **RC0** goes low for 2 cycles then high for 50 cycles. It will continue cycling until **INT1** goes high.

The last line generates a signal on

- •**RC3** that starts **low**,
- •stays low for **3 cycles**,
- •then goes high for **30 cycles**.
- •It starts this operation only after **1500 cycles** have elapsed since the program was **reset**, and
- •will stop **1800 cycles** after the program started, being active for only those 300 cycles.

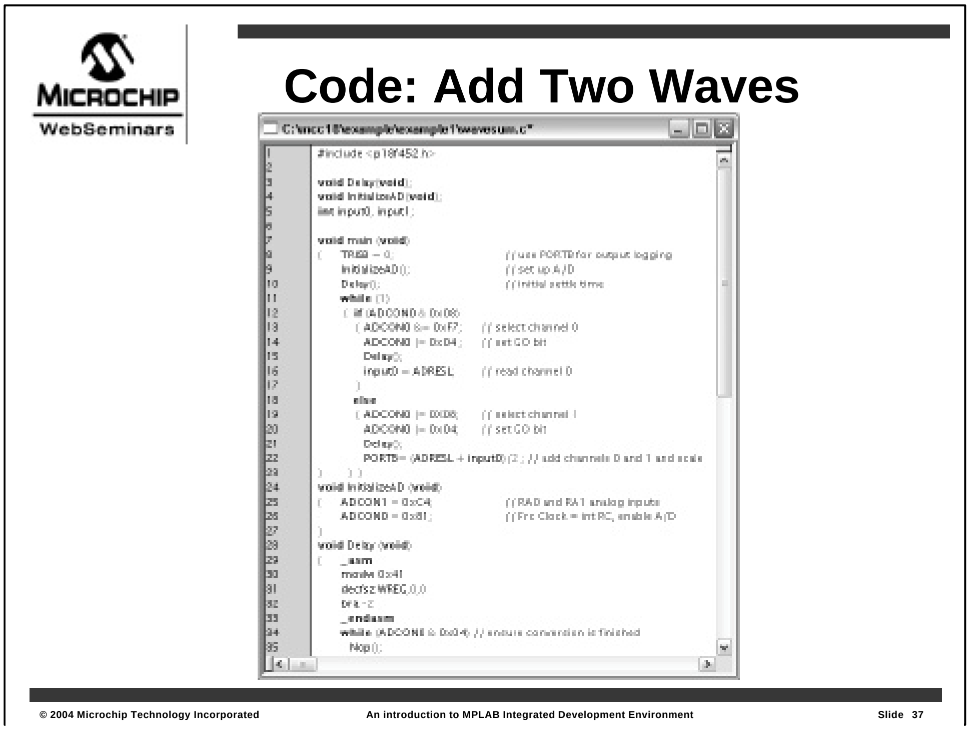
The last two tabs, Register Injection and Register Trace will be used in the following example.



For this next example, **two waveforms** will be **added** together and sent out **PORTB**, possibly to go to a Digital to Analog converter.

The diagram represents the **two waveforms** on the **left** being applied to to **A/D pins** of the **microcontroller** chip in the **middle**.

The **8 Pins** from **PORTB** go out to some device that will convert them back into analog signals.



This is the entire program that will be used to add the two waves that are coming in to the A/D converter from two pins, then output that sum to PORTB.

MICROCHIP WebSeminars

Add Two Waves Code

```
while (1)
{ if(ADCON0 & 0x08)
     { ADCON0 &= 0xF7;   // select AN0
       ADCON0 |= 0x04 ;  // set GO bit
       Delay();
       input0 = ADRESL;  // read AN0
     }
  else
     { ADCON0 |= 0X08;   // select AN1
       ADCON0 |= 0x04;   // set GO bit
       Delay();
       PORTB = (ADRESL + input0)/2;
                    // add AN0 AN1 and scale
     }
}
```

 An introduction to MPLAB Integrated Development Environment Slide 38

Here is just the essential code from the main routine.

The code executes in this infinite loop, alternately getting inputs from pins AN0 and AN1.

Line 2 looks at **bit 3** of **ADCON0** to see whether **AN0** or **AN1** was used in the last conversion, and switches to the alternate input pin.

The **A/D conversion** is started by

- setting **bit 3**, the **GO** bit in ADCON0,
- then a **delay** is needed to allow conversion time before reading.

The first wave is read in from **AN0**, and stored in the variable named **input0**.

Add Two Waves Code

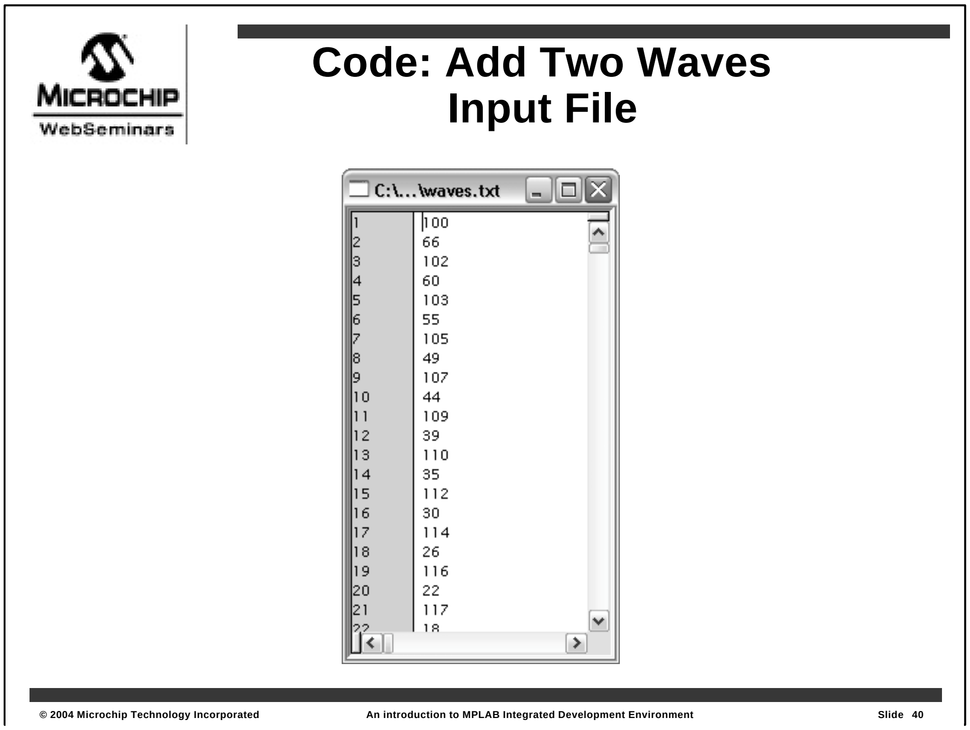
```
while (1)
{ if(ADCON0 & 0x08)
     { ADCON0 &= 0xF7;   // select AN0
       ADCON0 |= 0x04 ;  // set GO bit
       Delay();
       input0 = ADRESL;  // read AN0
     }
  else
     { ADCON0 |= 0X08;   // select AN1
       ADCON0 |= 0x04;   // set GO bit
       Delay();
       PORTB = (ADRESL + input0)/2;
                    // add AN0 AN1 and scale
     }
}
```


The next time through the loop, the “else” clause will execute, and

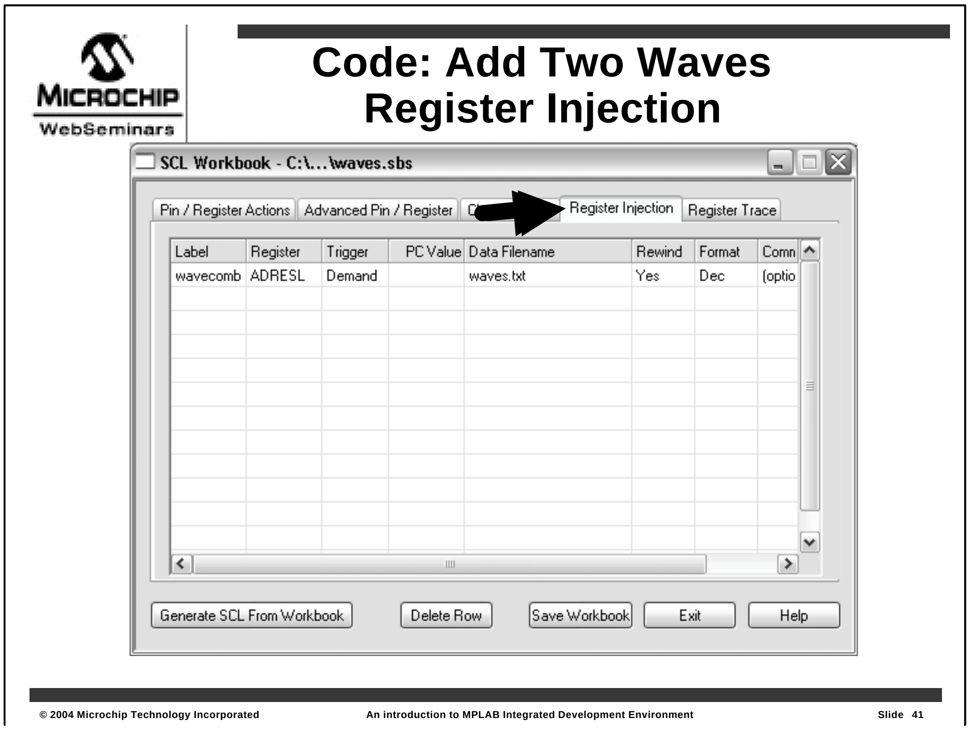
•**input0** will be added to

•the A/D signal from **AN1** and

•sent out **PORTB**.



A **source file** is made up containing the values for the waves being applied to the two A/D pins

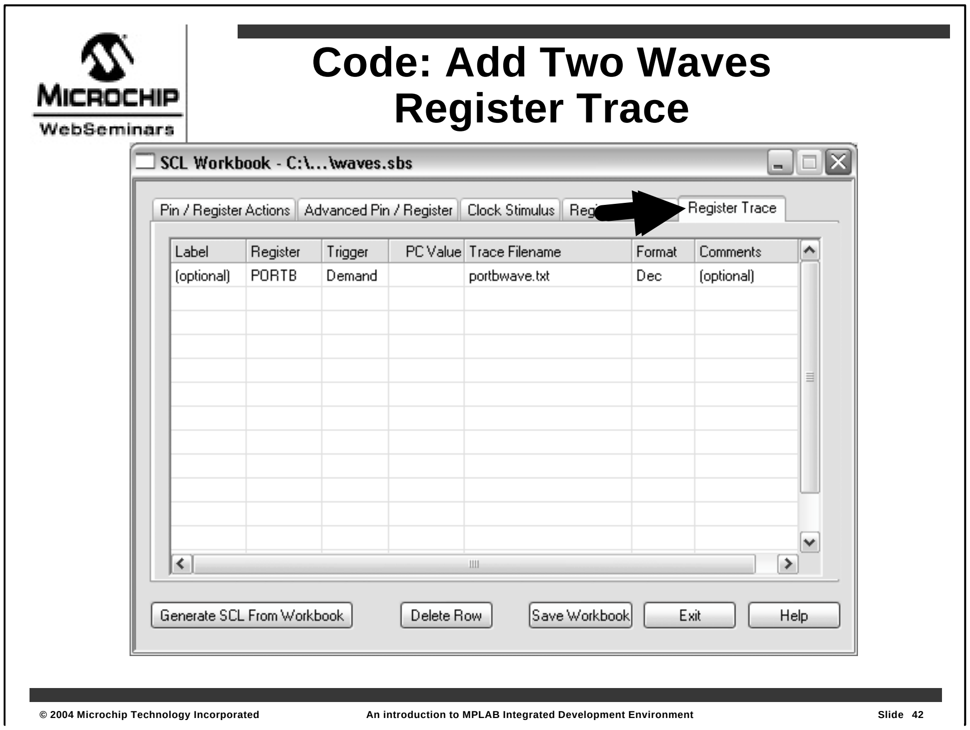


The register injection tab of the SCL workbook allows us to set up the **A/D conversion register** to receive it's input from the **data file**, named **waves.txt**.

The number will be

- interpreted as **decimal** values,
- will rewind –starting over when the end of the file is reached–
- and the values will be sent to the A/D conversion register **on demand**.

This means that each time the A/D conversion register is read by our program, the next value in the file will be used.

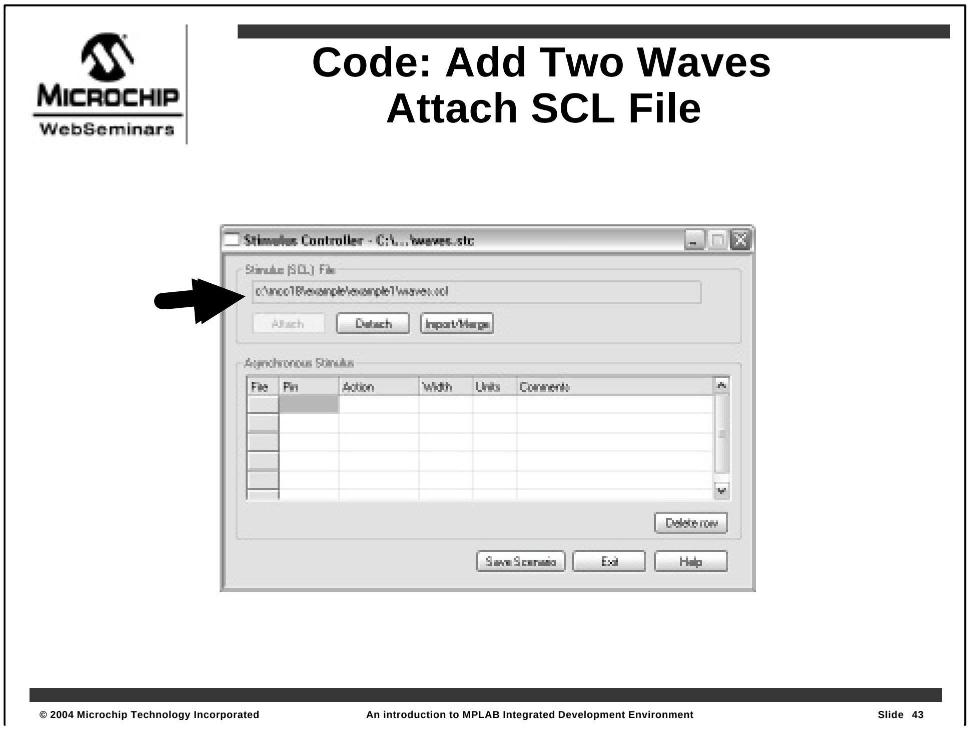


In order to validate that the code is working, we use the **Register Trace** tab of the SCL workbook

and set **PORTB** to be logged **on demand**, in other words, each time it is **written** to.

The values will be sent as decimal number to the file named **PORTWAVES.TXT**.

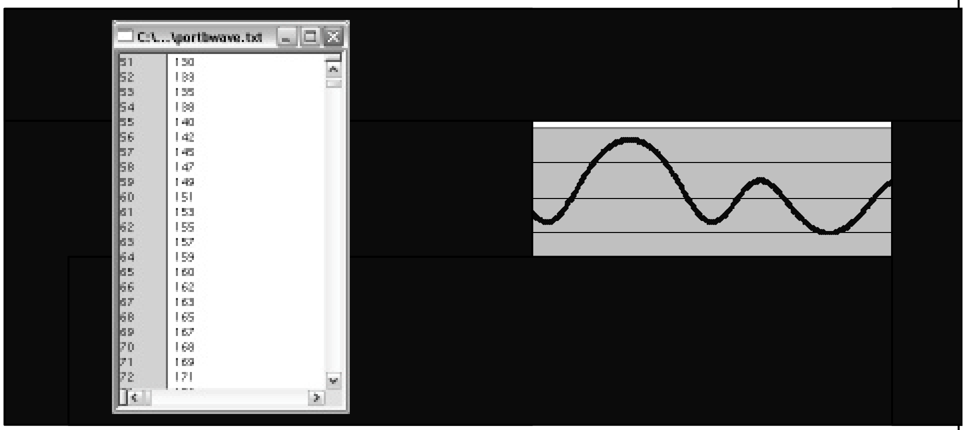
Now we generate the **SCL file** by **compiling**…<click>



…then open the **Stimulus Controller** and attach our SCL files.

Now we can **run** the application.

Code: Add Two Waves
Graph Output



 An introduction to MPLAB Integrated Development Environment Slide 44

After stopping the application, we have the results of our log in the file named **PORTWAVES.Txt**.

We can open the file to look at it, then paste the values into a program like **MATLAB** or **Excel** to **graph** the wave.

MPLAB® SIM Technology

dsPIC30F

- High Speed
- Many Peripherals Simulated
- Complex Stimulus and Output File Logging
- MPLAB C30 C Compiler `printf()` Support

PIC18

- High Speed
- Many Peripherals Simulated
- Complex Stimulus and Output File Logging
- MPLAB C18 C Compiler `printf()` Support **SOON!**

PIC10/12/16

- High Speed **SOON!**
- Many Peripherals Simulated **SOON!**
- Complex Stimulus and Output File Logging **SOON!**

 An introduction to MPLAB Integrated Development Environment Slide 45

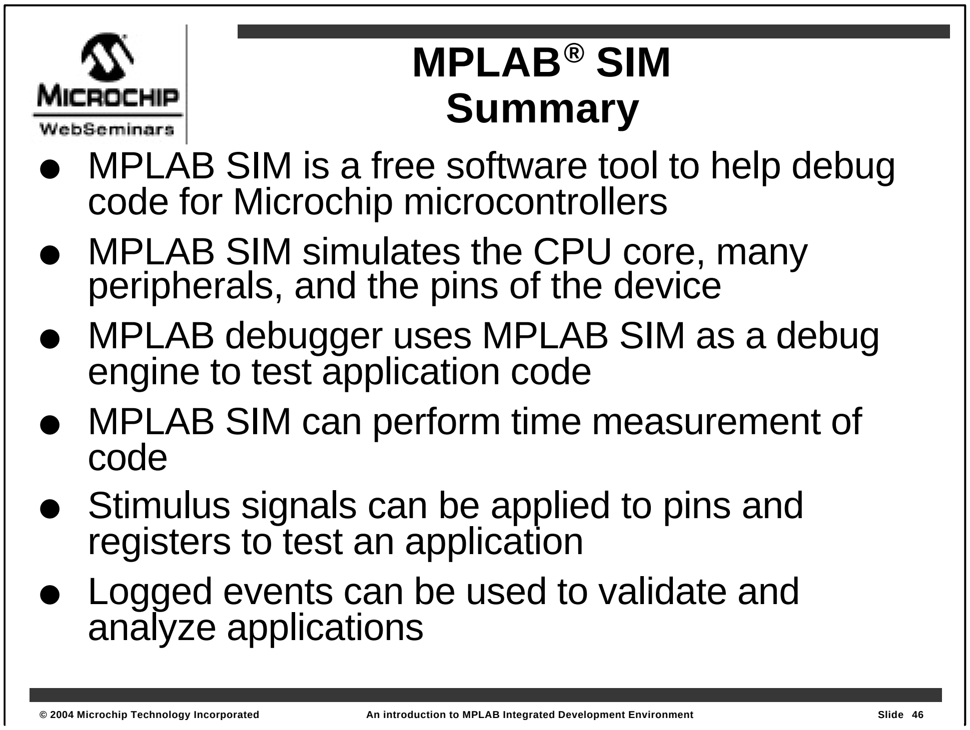
This has been a short demonstration of some of the powers of **MPLAB SIM** in conjunction with the C compilers, MPLAB C30 and MPLAB C18..

Currently the high speed and extensive peripheral simulation extend to the **PIC18** and **dsPIC** microcontrollers. The **dsPIC** family has one additional stimulus feature: the ability to use **`printf()`** statements from **MPLAB C30** to log output files. With **`printf()`** logging, reports can be generated with **precise formatting**, and with **complex data types**.

All Microchip Technology microcontrollers are simulated in MPLAB, but only the **PIC18** and **dsPIC** devices have these **complex stimulus features**, **high simulation speed** and **extensive peripheral simulation**.

As this web seminar is being developed, **MPLAB v6.60** is the latest release. This **new simulation technology** is being tested in the **PIC10/12/16** series microcontrollers, and the **`printf()`** feature is being added to **MPLAB C18.**

Be prepared to see the simulator get turbo-charged when these are implemented in future versions of MPLAB IDE and the MPLAB C18 compiler.



To recap the features of MPLAB SIM:

MPLAB SIM and MPLAB are available **free** for download from **www.microchip.com**.

It supports all current PICmicros and dsPIC microcontrollers made by Microchip Technology, and as new processors are developed, MPLAB SIM is extended to support them.

MPLAB SIM simulates the CPU core, the various memory areas, many peripherals and the pins of the microcontroller.

MPLAB's debugger functions use MPLAB as one of several debug engines to test code. The other debug engines are in-circuit emulators and in-circuit debuggers.

MPLAB SIM can accurately measure code timing using the Stopwatch or the time stamp feature of the Trace Buffer.

Stimulus signals can be applied to pins and registers. These signals can be triggered manually, can be set up as lists of events, can be repetitive waveforms, and can be activated by complex conditions.

And did we say that you can download **MPLAB IDE** which includes **MPLAB SIM** and all these features **free** from the Microchip Web site?

Thank you for your time.